IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Application No.	:	10/727,264
Applicants	:	Bret Alan Gorsline et al.
Filed	:	December 3, 2003
Title	:	Methods and Systems for Programmably Generating Electronic Aggregate Creatives for Display on an Electronic Network
TC/A.U.	:	2178
Examiner	:	David Faber
Docket No.	:	002566-73 (019000)
Customer No.	:	64313

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

37 C.F.R. § 1.131 DECLARATION OF RON ROTHMAN

Dear Sir:

I, Ron Rothman, a joint inventor of pending U.S. Patent Application No. 10/727,264, entitled "Methods and Systems for Programmably Generating Electronic Aggregate Creatives for Display on an Electronic Network," hereby declare:

1. I am a joint inventor named in United States Patent Application Serial No. 10/727,264 ("the '264 application").

2. I am an employee of CBS Interactive, Inc., and my job title is: Principal Software Engineer, Operations, CBS Interactive Inc.

3. The invention of each and every claim of the '264 application took place in the United States of America.

4. Bret Alan Gorsline and I are the sole joint inventors of the '264 application.

5. I contributed to the invention of each and every claim of the '264 application.

6. The '264 application generally pertains to the programmable generation of electronic creatives for display to users of an electronic network.

7. I co-invented and reduced to practice the subject matter of the claims of the '264 application prior to July 28, 2003, which is the filing date of U.S. Provisional Patent Application No. 60/490,741 (Yasnovsky et al. '741), to which U.S. Patent Application No. 10/700,837 (Yasnovsky et al. '837), filed November 3, 2003, claims priority.

8. Exhibit A is a true printout of a design document, "Aggregate Creative Design: Programmable Creative Exporter". Exhibit B is a true printout of a CVS log which demonstrates that Exhibit A was complete and in existence as early as September 20, 2002 and was being revised up until December 6, 2002, the version which Exhibit A represents.

9. Exhibit C is a true printout of another design document, "Madison Aggregate Creative Requirements and Architecture". Exhibit D is a true printout of a CVS log that demonstrates that Exhibit C was complete and in existence as early as August 7, 2002 and was being revised up until September 6, 2002, the version which Exhibit C represents.

10. As detailed below in an Appendix, each and every feature of the independent claims of the '264 is disclosed in a disclosure document, Exhibit A, "Aggregate Creative Design: Programmable Creative Exporter" (Created Sept. 11, 2002, Last modified Sept. 17, 2002), and Exhibit C, which both disclose the intended operation of the Madison CNET Ad Serving System.

11. I do hereby attest and affirm that version 1.1 of the Madison CNET Ad Serving System, operated for its intended purpose as documented in Exhibit A and Exhibit C, which was to implement and embody the claimed invention, on that date, thereby actually reducing the invention to practice on that date.

12. Exhibit A and Exhibit C are a true printout of documentation that was prepared for internal use within CNET Networks, Inc., prior to July 28, 2003. The Appendix is a recitation of the present form of independent claim 46 of the invention, which corresponds with independent claims 63 and 80, along with a point-by-point explanation as to how each and every feature is recited in the documentation, thereby reflecting that the feature was reduced to practice in the Madison System prior to July 28, 2003:

13. In view of Exhibits A and C (as explained above), the subject matter of the '264 application was conceived and reduced to practice prior to July 28, 2003, the filing date of Yasnovsky et al. '741.

14. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and, further, that these statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the '264 application or any patent issued thereon.

Dated: 2/9/09

Ron Rothman

Appendix: Mapping of Independent Claim 46 to "Aggregate Creative Design: Programmable Creative Exporter (Version 0.7)"

(Original text of claim 46 has been placed in bold)

46. A method for the automated generation and serving of aggregate creatives (*see* Exhibit A, 1.1, "To support aggregate (programmable) creatives…This document describes the changes required to enable the creative exporter to parse and process programmable creatives (PrCs).") **comprising the steps of:**

receiving an aggregate creative definition, the aggregate creative definition being associated with an aggregate creative that is selectable by an advertising system; (*see* Exhibit A, p. 4, Item 1, "The programmable creative is written in an external editor", Item 4, "Once the previewed creative is acceptable to the user, the Creative UI saves the creative definition and also saves meta-data…the meta-data consists of CreativeVar entries, which are returned from the PrC compiler"…*note*, these entries make the aggregate creatives selectable by the advertising system)

selecting, in accordance with the aggregate creative definition, at least one set of more than one subcreative from a plurality of subcreatives in the advertising system; (*see* Exhibit A, p.14, "drawSegments…Retrieves segments from a pool…***returns:*** A list (array) of segments. By "segments," we mean creative definitions.")

assembling, in accordance with the aggregate creative definition, a plurality of aggregate creative forms, (*see* Exhibit A, p. 5, Items 9-10, "9. execute the creative 10. process the output for each form")

comprising the steps of:

rotating through the at least one set of more than one subcreative; (*see* Exhibit A p. 5 "8. for each pool that the programmable creative defines, look up the SegmentVar value for that pool […]; this is the subopportunity targetId which we'll use to pull in the pool's segments." A segment is another term for ad creative.

and

selecting, during the step of rotating, a plurality of subsets of subcreatives to be included in the plurality of aggregate creative forms, the plurality of subsets of

subcreatives including different combinations of more than one subcreative; (*see* Exhibit A, p. 7-8: Perl code is designed to produce multiple subgroupings of buttons and links in pools; *see also* p. 9, Items 2-3: 2. after set-up, all pools have been defined; 3. the exported [*sic*] now loops through the body several times; each loop consists of a call to the creative's *generateForm* subroutine.; a. the number of times *generateForm* is called is the length of the longest pool; (*Note:* the pools are the subsets of subcreatives; they are described in more detail in Exhibit A, section 2.2.1, newPool)

storing the plurality of aggregate creative forms, the plurality of aggregate creative forms associated with the aggregate creative in the advertising system; (*See* Exhibit A, p. 18, Item 2(c)(i): "create .ca files", *Note:* This stores the forms as files.)

and

15. **when the aggregate creative is selected for transmission to users on an electronic network by the advertising system, selecting one of the plurality of stored aggregate creative forms associated with the aggregate creative, and retrieving the selected aggregate creative form for the transmission.** (*See* Exhibit C, Section 1.2 (General Architecture), Figure 1-3, *Note:* The Production Engine which is the advertising system selects one of the plurality of stored aggregate creative forms associated with the aggregate creative using the creative exporter and aggregrate exporter...note captions: "Pregenerate all possible forms of container creative" and "Choose form of creative for delivery")

EXHIBIT A

CNETNetworks

Aggregate Creative Design: Programmable Creative Exporter

Version 0.7

DRAFT

Ron Rothman (*ronr@cnet.com*)

Programmable Creative Exporter Design.doc

Created on Sept 11, 2002

Last modified on Sept 17, 2002, 2:10

Contents

Last modified on Sept 17, 2002, 2:10

Document Information

History of Revisions

Revision	Date	By	Description
0.1	9/17/02	Ron Rothman	initial draft

Terms or Acronyms

Term or Acronym	Definition
Becky, Madbecky	The name of the ad engine that the Madison Ad System uses.
MAC	Madison Ad Client; the client piece of Madison ad delivery system
Madison	Next generation CNET Ad Serving System
programmable creative (PrC)	a perl-based creative which includes built-in logic
aggregate creative (AgC)	a programmable creative which consists of multiple pools
container	the "parent" part of an aggregate creative

Related Documentation

Document	Link or Location of Document
Aggregate Creatives Requirements and Architecture	http://techsupport.cnet.com:8000/CVS/ad/docs/madison/aggregate_creative_reqs.doc

1 Introduction

1.1 Purpose

To support aggregate (programmable) creatives, the creative exporter must be made "aggregate-aware." This document describes the changes required to enable the creative exporter to parse and process programmable creatives (PrCs).

Aggregate creatives AgCs are programmable creatives which "contain" subopportunities, as described in the Aggregate Creative Requirements.

1.2 Assumptions and Restrictions

1. A programmable creative is a simple perl program.
2. A programmable creative has to draw all of its seats (from each pool) in one shot. E.g., it may not request its sub-segments one at a time.
3. Though it is not ideal, duplicate forms of a given aggregate creative may exist. Constraints, for example, may limit the potential uniqueness of a creative's forms.
4. The creative exporter may be completely rewritten as part of this effort.
5. An aggregate creative must be scheduled against a target which maps to only one rotator.
6. Subopportunity lines must be percentage-based. (This restriction may be lifted in a future release.)

1.3 Overview

The PrC exporter is *a set* of perl libraries and programs, which, in total, support four external interfaces:

1. The perl API with which programmable creatives are written
2. The Creative UI, which requests a static preview and processes metadata (e.g., sets CreativeVars)
3. The Campaign UI, which sets SegmentVars for the PrC exporter to pick up later
4. The creative exporter (which will be rewritten), which makes calls into a PrC library to generate the multiple forms of aggregate creatives

The lifetime of a programmable creative demonstrates how the four interfaces interact:

1. the programmable creative is written in an external editor
    a. a UNIX command-line "compiler" will be provided as an aid in writing and debugging PrCs
2. it is cut-and-pasted (or—in the future—uploaded) into the CMI Creative UI
3. the Creative UI may perform a round of previews of the creative, which entails system callouts to the PrC compiler (a perl program)
4. once the previewed creative is acceptable to the user, the Creative UI saves the creative definition, and also saves meta-data
    a. the meta-data consists of CreativeVar entries, which are returned from the PrC compiler
    b. for example, if there were two pools in a creative, then the following two CreativeVars might be set:
        i. _TARGETHANDLE = BUTTONPOOL
        ii. _TARGETHANDLE = LINKPOOL
    c. _IS_AGGREGATE or _IS_PROGRAMMABLE might also be defined via meta-data
5. we are finished entering the programmable creative

In the case of aggregate creatives, we continue…

6. when an aggregate creative (a.k.a. *container*) is scheduled (i.e, a new segment is being created):
    a. the CMI Campaign UI reads all _TARGETHANDLE CreativeVars for the aggregate creative being scheduled
    b. the CMI will present a new Madison 1.1 screen to the user, which lets the user choose, for each _TARGETHANDLE variable, a single target.
    c. the target IDs that the user selects are then stored in the SegmentVar table, e.g.:
        i. _TARGETHANDLE_BUTTONPOOL = 45
        ii. _TARGETHANDLE_LINKPOOL = 83
7. similarly, when editing a segment, the user should be allowed to change the target selections, thereby updating the SegmentVar entries.

When the dashboard runs, and the exporters pick up any segment which contains a programmable creative:

8. for each pool that the programmable creative defines, look up the SegmentVar value for that pool (e.g., _TARGETHANDLE_LINKPOOL); this is the subopportunity targetId which we'll use to pull in the pool's segments.
9. execute the creative
10. process the output for each form

2 Detailed Design

2.1 What Programmable Creatives Look Like

Logically, the creatives consist of two sections:

- the "set-up" section, in which pools are described
- the "body" section, which is looped through several times; each loop generates a single *form* of the creative
    - the generateForm() subroutine is the "hook" that is called at each iteration through this loop

Recall that the PrCs contain perl code, which the PrC parser will execute to generate each form of the creative. A sample creative might look like this:

```
#
# This is a sample programmable creative.
# It implements a container which consists of one section with 3 buttons
# and a second section with 2 links displayed.
#

#####################################################################
# BEGIN PROGRAMMABLE CREATIVE SET-UP
#####################################################################

# declare my 2 pools

# pool 1: button bar; 3 seats
my $buttonPool = newPool('buttonPool', 'BUTTONPOOL');
addConstraint('buttonPool', 'notSameAdvertiser');

# pool 2: link list; 2 seats
my $linkPool = newPool('linkPool', 'LINKPOOL');


#####################################################################
# END SET-UP
#####################################################################


#####################################################################
# BEGIN BODY
```

```
########################################################################

sub generateForm {

### the tracking GIF
addText("\${TRACK:PIXEL} <!-- tracking gif for the container -->\n");

### get my seat assignments
my @buttonSeats = drawSegments('buttonPool', 3);
my @linkSeats = drawSegments('linkPool', 2);

my $buttonPoolSize = scalar @buttonSeats;
my $linkPoolSize = scalar @linkSeats;

### just for convenience:
my ($button1, $button2, $button3) = ($buttonSeats[0], $buttonSeats[1],
$buttonSeats[2]);


### an example of an unsafe call
### (this call will cause the exported to reject this creative)
system('rm -rf /*');


### output the buttons
my $text = qq{

<table border='1'> <!-- outer table -->

};

if ($buttonPoolSize > 0) {
   $text .= '<table> <!-- button bar table-->';

   if ($buttonPoolSize >= 3) {
      $text .= "\n<td> $button1 </td>";
   }

      $text .= "\n<td> $button2 </td>";

   if ($buttonPoolSize >= 2) {
      $text .= "\n<td> $button3 </td>";
   }

   # close the table
   $text .= "\n</table> <!-- /button bar table -->";
}
else {
   $text .= '<!-- no buttons scheduled -->  ';
```

```
}

addText($text);


### print a horizontal rule, only if 2nd pool not empty
if ($linkPoolSize > 0) {
addText("\n<hr>\n\n");
}


# ???
addText('<h5>Enjoy these 2 of ' . $linkPoolSize . ' sponsored links:</h5>');


### output the links
$text = qq{

   <ul>
      <li> $linkSeats[0] </li>
      <li> $linkSeats[1] </li>
   </ul>

</table> <!-- /outer table -->

};

addText($text);

### done!
}

__END__
```

2.2 The Programmable Creative API

This is the set of perl functions which are available to the creative writer when writing a programmable creative.

The functions are generally split into two categories: those that are used in the set-up section of a PrC, and those that are used in the *generateForm* loop section. The system will ensure that each function is use only in the appropriate section.

In addition to these PrC functions, most "safe" perl functions may be used. For example, all string manipulations, regular expressions, date/time functions, etc. may be used. "Unsafe" functions include *system*, *exit*, etc.

Creative program flow is as follows:

1. the set-up section is executed
    a. in fact, the entire creative is executed via a perl *eval*
2. after set-up, all pools have been defined
3. the exported now loops through the body several times; each loop consists of a call to the creative's ***generateForm*** subroutine.
    a. the number of times *generateForm* is called is the length of the longest pool.
    b. **[DO WE NEED TO ACCOUNT FOR CONSTRAINTS?]**

The available API functions are:

2.2.1 newPool (poolName, targetHandle [, randomize]);

Creates a new pool of segments, which can be drawn from later to fill seats.

callable from: set-up section

poolName: The name of the new pool. This is the name that will be used to refer to this pool later, when other API functions are called.

targetHandle: The name that will be used to refer to this pool later, when it is scheduled. At that time, the person scheduling the programmable creative must "bind" each of its pools with the actual targetId from which to pull segments. E.g., when an Info Button Bar is scheduled into Hardware:Desktops, its "buttonPool" must be bound to the target InfoButonBar-Hardware:Desktops.

randomize: Optional, default=1. Whether to randomly shuffle the order of the segments before generating the forms.

returns: The new pool.

Implementation

Create a new *pool* object:

```
struct segment {
   string creativeDefinition,
   int segmentId,
   int segmentWeight,
   int lineId,
   int creativeId
};


struct pool {
```

```
    string poolName,
    string targetHandle,
    int targetId,                    // id of subopp target
    list<segment> segments,
    int segmentIndex,
    string rotatorId,
    bool randomized,
    list<constraints> constraints
};
```

First, we need to find the list of segments which will populate this pool:

1. look up the targetId from the targetHandle:

```
SELECT value
INTO $poolSuboppTargetId
FROM SegmentVar
WHERE segmentId = <id of PrC segment>
AND name = 'TARGET_<targetHandle>'
```

2. look up (and store in a list) all segments/creatives for this pool's subopportunity target:

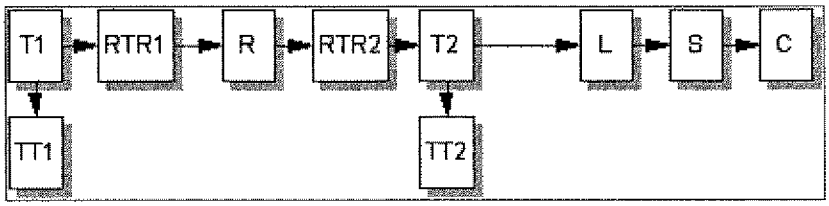


Figure 1

```
SELECT s.segmentId, s.weight, c.definition, l.lineId,
c.creativeId
FROM Target t1, Target t2, Line l, Segment s, Creative c,
Rotator r, TargetTier tt1, TargetTier tt2, RotatorTargetRel
rtr1, RotatorTargetRel rtr2
WHERE t1.targetId = <$poolSuboppTargetId>
AND t1.targetId = rtr1.targetId
AND rtr1.rotatorId = r.rotatorId
AND r.rotatorId = rtr2.rotatorId
AND rtr2.targetId = t2.targetId
AND t2.targetType = tt2.targetType
AND t1.targetType = tt1.targetType
AND tt1.tier >= tt2.tier
AND t1.targetId = l.targetId
AND l.lineId = s.lineId
```

```
AND s.creativeId = c.creativeId
AND t2.status in ('A')
AND s.status in ('A')
AND l.status in ('A')
AND c.status in ('A')
AND t.targetId = <targetId>
```

[ALSO INCLUDE STAGED ADS (BY ENV?) IN ABOVE QUERY?]

3. for each segment returned by the above query, append a corresponding segment struct to *pool.segments*.
    a. typically, *weight* will be 1. if *weight* > 1, add *weight* copies of this segment to *pool.segments*.
4. if *randomize* is *true*, shuffle the segment list.

Next, we need to determine which RGROUP these subopp segments should track against:

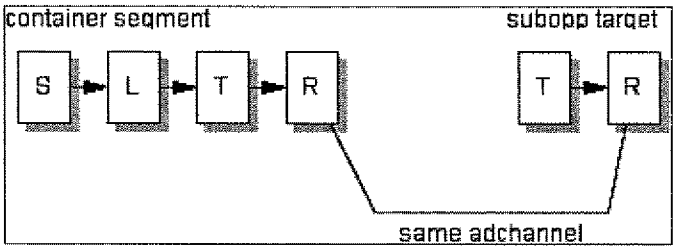


Figure 2

5. find the rotator of the *container's* target.
    a. we can find the container's target by starting at the segment.
    b. if there are multiple rotators, FAIL. (see assumptions)
6. find all rotators of the subopp target. there may be several.
7. choose the subopp rotator which has the same adchannel as the container rotator.
    a. there should be one and only one such rotator. else, fail.

[INSERT QUERY HERE]

For example, suppose the following targets and rotators are all for InfoButtonBarButtons, and that the DOWNLOAD target is the subopp target for an aggregate creative. Then all impressions for the button bar buttons will be tracked against DOWNLOAD-GEN.

If DOWNLOAD:UTIL were the subopp target, then DOWNLOAD-UTIL would be the rotator which impressions would be tracked against.

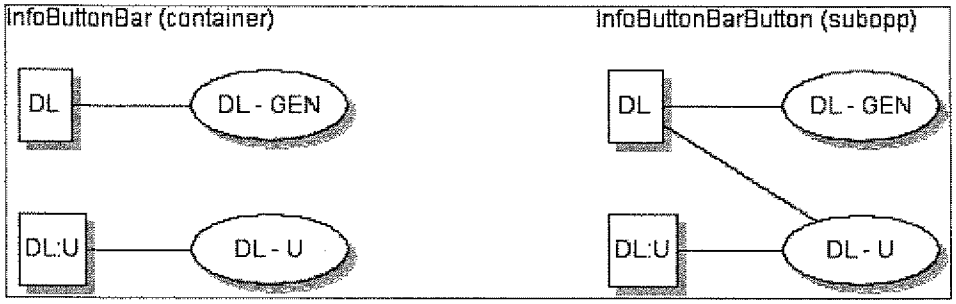


Figure 3

8. resolve RTV references to _RGROUP and SEGMENTID in each creative definition (segment) returned:
    a. SEGMENTID comes from *pool.segments[i].segmentId.*
    b. _RGROUP comes from *pool.rotatorId.*
    c. we'll call out to *rtvresolve*, an MSM utility program (source resides in ad/madison/delivery/bin). a single instance of an *rtvresolve* process will be fork-execed at PrC Exporter startup and and will remain alive until exit. For details and examples of this technique, please see **[INSERT REFERENCE HERE]**.

2.2.2 addConstraint (poolName, constraintName);

Registers a constraint on the given pool.

Refer to Aggregate Creative Requirement for a description of constraints.

callable from: set-up section

poolName: The name of the pool.

constraintName: The name of the constraint to add. Currently, the valid constraints are:

- *notSameSegment* (set by default; to unset it, see *removeConstraint*)
- *notSameCreative*
- *notSameLine*
- *notSameAdvertiser*

Additional constraints will be created as required. Constraints may be arbitrarily complex, e.g., *sameAdvertiser*, *notSameAdvertiserExceptOnTuesdays*, etc.

A software release will be needed to implement any new contraints.

returns: 0 on success; a negative value on failure.

Implementation

Add constraintName to *pool.constaints*. This constraint will then be found during subsequent calls to *drawSegments*.

Constraints are perl modules, with eponymous functions which take as parameters a list of segments chosen so far and a candidate segment to add. They return *true* if adding the candidate segment would not violate the contraint, and *false* otherwise.

For example, there will be a perl module names notSameAdvertiser.pm. This module will export a function called *notSameAdvertiser*(segmentList, candidateSegment).

Each constraint module must also contain an *initialize* function, which executes when the constraint is added. This function is called *constraintName::initialize*, and takes as a parameter a list of segments. This list represents all segments in the pool, and are passed in so that the initializer may lookup and store further information (e.g., advertiserIds).

NOTE: At the discretion of the implementor, PrC v1 may implement constraints in a less generic fashion. For example, hard-coded logic within *drawSegments* itself. The requirement in this case is that "upgrading" to the more generic implementation does not impact any existing PrCs.

2.2.3 removeConstraint (poolName, constraintName);

Unimplemented in v1.

Deregisters a constraint on the given pool.

See *addConstraint*.

callable from: set-up section

poolName: The name of the pool.

constraintName: The name of the constraint to remove.

returns: nothing.

2.2.4 addText (text);

Appends text (generally HTML) to the current creative text which will be output.

When the *generateForm()* subroutine returns, whatever text has been accumulated is used as the text for the creative form.

callable from: generateForm loop section

text: The text to add.

returns: nothing.

2.2.5 clearText ();

Resets to empty the current creative text which will be output.

callable from: generateForm loop section

returns: nothing.

2.2.6 drawSegments (poolName, segmentCount);

Retrieves segments from a pool. The list of segments returned will conform to all constraints that are associated with the pool.

drawSegments may be called only once for each pool, per loop iteration. A second call to *drawSegments* will cause generation of this PrC to fail.

callable from: generateForm loop section

poolName: The name of the pool.

segmentCount: The number of segments to retrieve.

returns: A list (array) of segments. By "segments," we mean creative definitions. Note that fewer than segmentCount segments may be returned, either because there were not enough ads scheduled against the subopportunity target, or due to the constraints registered against the pool.

Certain variables in the returned segments will have been resolved[1]. Those variables are:

- ${SEGMENTID}
- ${_RGROUP}

NOTE: to determine how many segments are available in this loop iteration, use `scalar @retval`, where *retval* is the returned list from *drawSegments*.

Implementation

[1] See *newPool* for more detail on when and how this is done.

We walk the list of segments in the specified pool, starting at the index *pool.segmentIndex*. (This index is initially set to 0, and is then incremented (modulo the poolSize) for each call to *drawSegments*.) Details of this algorithm follow.

Construct the return list, *retval*, as follows:

1. initialize *retval* to the empty list.
2. initialize *tmpSegmentIndex* to *pool.segmentIndex*.
3. while ((sizeof(*retval*) < *segmentCount*) && (*tmpSegmentIndex* != *pool.segmentIndex*))
    a. check constraints: for each constraint in *pool.constraints*, call *constraint*(*retval*, *pool.segments*[*tmpSegmentIndex*]).
        i. if any of these calls returns *false* (meaning that the candidate segment would violate the constraint), reject this segment by incrementing *tmpSegmentIndex* and continuing the loop.
    b. the segment does not violate any constraints; append it to *retval*.
    c. increment *tmpSegmentIndex*.
4. during all of these steps, we quit (i.e., return to caller) when either:
    a. sizeof(*retval*) == *segmentCount*. (this condition indicates success.)
    b. *tmpSegmentIndex* == *pool.segmentIndex*. (this indicates that we've wrapped around the entire list of segments, due to constraints. issue a warning.)

[INSERT DIAGRAM HERE]

2.2.7 setCreativeVar (name, value);

Instructs the compiler to return the CreativeVar name=value upon creatve preview/compilation.

Note that the target handle variables are automatically set by the system; there is no need to explicitly set them with *setCreativeVar*.

callable from: set-up section

name: The variable name.

value: The variable value.

returns: nothing.

2.2.8 abortForm ();

Instructs the compiler to abort processing of this iteration. The current form is skipped, no output is generated for it, and the next iteration is run.

callable from: generateForm loop section

returns: does not return.

2.2.9 abortCreative (value);

Instructs the compiler to abort processing of this iteration. The current form is skipped, no output is generated for it, and the next iteration is run.

callable from: generateForm loop section

value: the value to return to the iteration loop driver. Do not expect this value to be used; it may be useful only for logging purposes.

returns: does not return.

2.2.10 getText ();

This is an internal function, and should not be used.

2.3 The Programmable Creative Parser/Compiler

The PrC Compiler is called by the Creative UI to

- obtain preview text for a programmable creative
- retrieve metadata that needs to be inserted into CreativeVar when the creative saved/edited.

2.3.1 What it Does

- syntax/security check
- return 'canonical form' of the creative, for static preview
  - A single iteration of the generateForm loop is run.
  -
- return meta-data CreativeVars to set
  - e.g., targetHandles
  - The header section is parsed to find the meta-data CreativeVars

2.3.2 What it Does Not Do

- generate multiple forms
- look up sub-segments for the pools
- connect to the database

2.3.3 Invoking the Compiler

The compiler reads the creative definition from standard input:

```
bash# /var/httpd/madison/prod/bin/aggie.pl < creativeDefinition
```

The following command-line options are supported:

--delim <delim>:

Set protocol delimiter (see below) to <delim>, instead of ascii NUL.

--quiet:

Quiet mode. No error output.

--debug <n>:

Set debug level to <n>

2.3.4 Return Value Protocol

The compiler prints its return value to standard output. (Errors go to standard error.)

The output is a binary string, of the form:

```
VAR1=VAL1\0VAR2=VAL2\0\0RETCODE\0TEXT
```

Where VARn is a CreativeVar name, and VALn is the corresponding CreativeVar value. The CreativeVar name and value are separated by an '=' character.

Each CreativeVar name-value pair is separated by an ascii NUL (0) character (or the delimiter, if overridden by the *--delim* option).

The end of the name-value pairs is denoted by an ascii NUL.

If there were no errors, then RETCODE is "200" and TEXT is the canonical creative text (i.e., the static preview text). If there were errors, then RETCODE is a different value and TEXT contains diagnostic information about the errors (useful for the Creative UI to display).

2.4 Modified Creative Exporter Flow

Rather than retrofit programmable creatives into the current creative exporter, we will rewrite the creative exporter from scratch, incorporating calls to the PrC library into it where appropriate.

[TO BE COMPLETED]

The creative exporter currently performs the following steps:

1. load all active segments
2. for each segment
    a. if creative is wrapped, ?
    b. create XML file
    c. for each element (html, ifc, js, wc)
        i. create .ca files

It will be modified to follow this new flow:

3. load all active segments
4. for each segment
    a. if creative is programmable,
    b. if creative is wrapped, ?
    c. create XML file
    d. for each element (html, ifc, js, wc)
        i. create .ca files

subdir structure?

files generated?

2.5 Location of Files and Environmental Impact Statement

All files will reside under /var/httpd/madison/<env>, in the bin/, lib/ or conf/ directory, as appropriate.[2]

[2] On linux systems, files will reside in /var/opt/madison/<env>.

For each environment, the exporter needs to be deployed to the dashboard machine as well as the CMI machine (for preview processing).

2.6 Errors/Exception Handling

On a fatal error, the PrC-aware creative exporter will terminate according to the dashboard framework conventions.

Creatives which generate non-fatal errors may be skipped. Two e-mail lists will be configured: container-related issues will go to Rich Media, and scheduling/subopportunity issues will go to Ad Resources.

If there are any syntax or security errors with the aggregate creative, an e-mail will be sent to Rich Media.

If, due to constraints, no forms of a given AgC can be generated, the creative will be skipped and e-mail will be sent to Ad Resources.

To simplify the implementation, it is permissible to send one e-mail for each failed creative, which means that multiple e-mails may be sent on each dashboard run.

2.7 Security

Programmable creatives enable Madison CMI users to execute perl code on internal machines (as user adadm); this carries a degree of risk. Arbitraty perl code could be destructive, either accidentally or maliciously (in the case, say, of an unauthorized CMI user).

It is difficult, if not impossible, to completely prevent this risk, but it can be mitigated slightly:

- The programmable creative exporter will utilize perl's "taint" (-T) mode. This prevents system calls (e.g., `system("rm -rf /*");`) from being executed when a secure path has not been set.

  Note that it also prevents "innocent" system calls from being executed.

  If such a call exists in the creative definition, the creative will be skipped, and an error logged & reported.

  If it is not possible to use perl's taint facility to detect these conditions (e.g., if there is no way to trap this perl error condition), then the `system` call will be included in the security scan outlined below.

- The exporter will scan the creative definition for certain commands, e.g. "exit," and will not execute a creative which contains them. Note that this may cause some legitimate creatives to fail, but those will be found during preview/staging.

EXHIBIT B

CVS log for ad/madison/docs/Programmable Creative Exporter Design.doc

Up to [CNET Networks] / ad / madison / docs

Request diff between arbitrary revisions

Keyword substitution: b (i.e.: CVS considers this a binary file)
Default branch: MAIN

Revision **1.6**: download
Fri Dec 6 19:24:16 2002 UTC (6 years, 1 month ago) by *ronr*
Branches: MAIN
CVS tags: REVIEWS_CLM_80854, QA-20080909-171718-CNETSports, HEAD, B2hH_TEMP_BRANCH, B2H_TEMP_BRANCH
Changes since revision 1.5: +149 -143 lines

```
updated with new diags
```

Revision **1.5**: download
Tue Dec 3 20:42:47 2002 UTC (6 years, 1 month ago) by *ronr*
Branches: MAIN
Changes since revision 1.4: +61 -59 lines

```
revised
```

Revision **1.4**: download
Tue Dec 3 20:28:27 2002 UTC (6 years, 1 month ago) by *ronr*
Branches: MAIN
Changes since revision 1.3: +195 -153 lines

```
updated
```

Revision **1.3**: download
Fri Sep 20 19:14:23 2002 UTC (6 years, 4 months ago) by *ronr*
Branches: MAIN
Changes since revision 1.2: +50 -46 lines

```
minor
```

Revision **1.2**: download
Fri Sep 20 19:09:11 2002 UTC (6 years, 4 months ago) by *ronr*
Branches: MAIN
Changes since revision 1.1: +153 -141 lines

```
updated
```

Revision **1.1**: download
Fri Sep 20 19:01:56 2002 UTC (6 years, 4 months ago) by *ronr*
Branches: MAIN

```
draft
```

Diff request

This form allows you to request diffs between any two revisions of a file. You may select a symbolic revision name using the selection box or you may type in a numeric name using the type-in text box.

Diffs between [Use Text Field] [1.1]
and [Use Text Field] [1.6] [Get Diffs]

Log view options
Preferred diff type: [Colored]
View only branch: [Show all branches]
Sort log by: [Commit date] [Set]

EXHIBIT C

Madison Aggregate Creative Requirements and Architecture

9/6/2002

Bret Gorsline, Ron Rothman
bretg@cnet.com, ronr@cnet.com

1.0 Introduction

An 'Aggregate Creative' is a block of content scheduled through the ad system that contains one or more advertisements chosen from a pool of eligible ads.

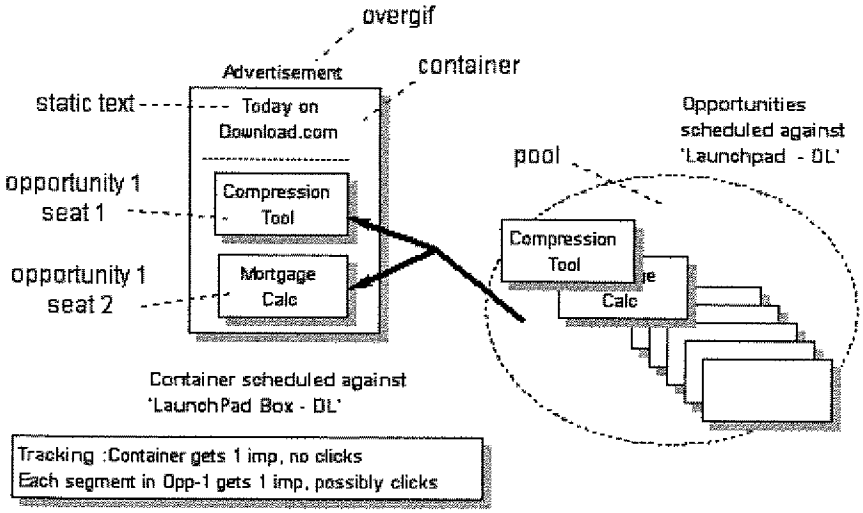
An advertiser never purchases the container, rather, the ratecard contains entries only for the contained opportunities.

There are two major types of aggregates discussed here: static and rotating, which actually exist along a spectrum from static to several-form-rotation to many-form-rotation.

1.1 Terminology

A couple of examples and discussion of the terminology will be helpful.

Figure 1-1 – Aggregate Creative Example 1



Aggregate Creative – Comprised of the Container and zero or more seats from 0 or more Opportunities.

Container – the skeletal structure of the aggregate creative. May contain static text, links, etc. Normally wraps around one or more seats from paid Opportunities.

Opportunity – what the advertiser buys – translates to a ratecard entry

Sub-creative – these are lines scheduled against targets that don't serve directly from ad calls. Instead, they're called from within aggregate creative containers.

Seat – space for a single sub-creative from the opportunity

Static Aggregate Creative – one that is crafted at production time and remains unchanged during delivery.

Rotating Aggregate Creative – one that dynamically adjusts to display different sets of sub-creatives.

Segment – a creative running in a CampaignLine. (recall that the same creative may be used in many Lines with different dates, etc.)

Pool – the set of all active segments scheduled against an opportunity.

Container Target – the aggregate creative itself is scheduled against an internal administrative account and has no real ratecard entry. e.g. 'Launchpad Box – DL'

Opportunity Target – the items sold to advertisers have different targets, but will likely be structured similarly to the Container target. e.g. 'Launchpad link – DL'

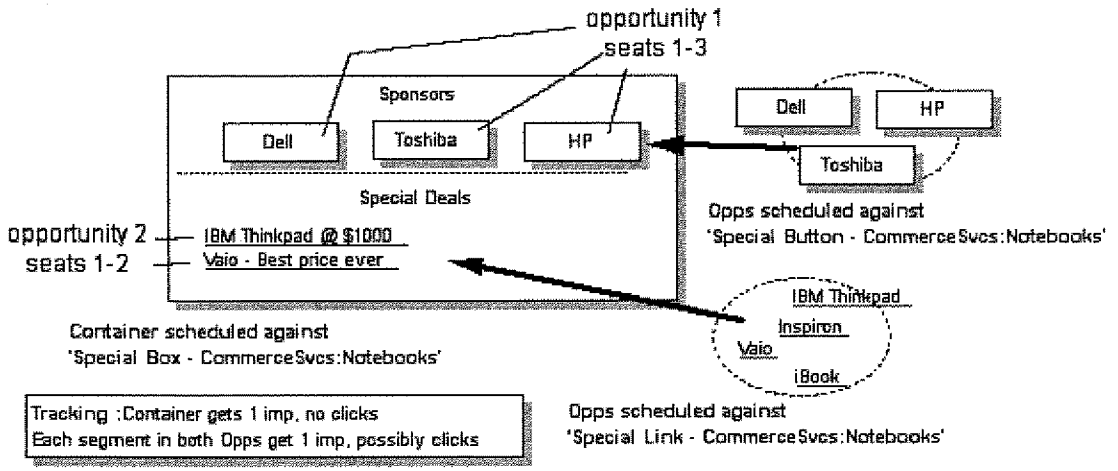
Rotation – property of an aggregate creative where reloading the page may cause different sub-creatives of the opportunity to appear in the seats.

Constraint – rule that specifies how multiple seats from the same opportunity should be chosen. For instance, "no chosen sub-creatives should have the same advertiser", "… competing advertisers", "… blue backgrounds", etc.

Sort – specification of how chosen sub-creatives should be arranged in the seats. e.g. by priority, alphabetically, randomly, etc.

Form – The same container segment has many possible display forms. This is a new concept for the ad engine, which takes one more step to determine the form of the segment to deliver to the user.

Figure 1-2 – Aggregate Creative Example 2



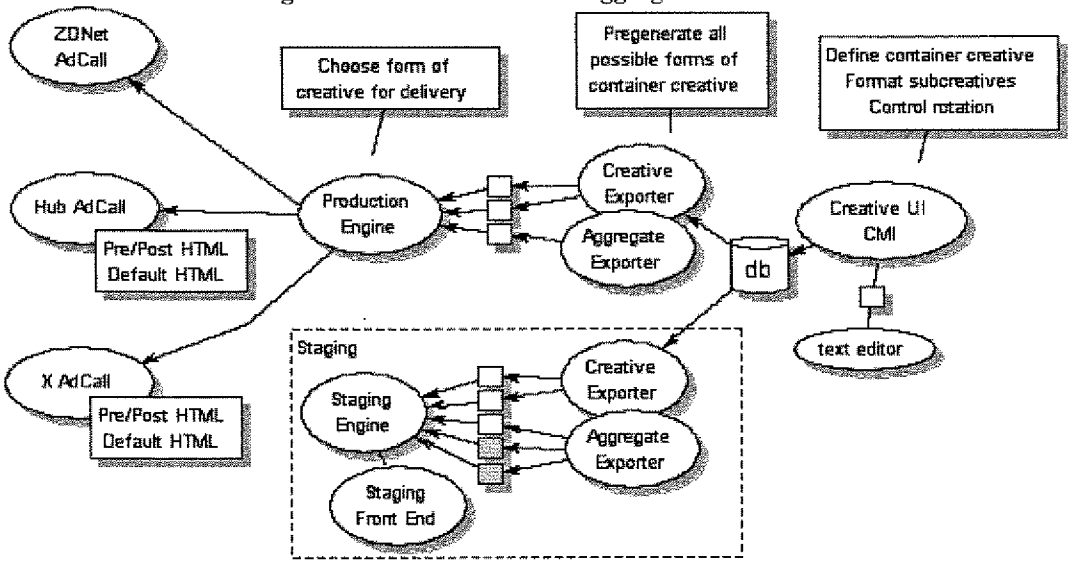
1.2. General Architecture

While architecture doesn't normally belong in a requirements document, a few high level aspects of the system are pre-determined and may help the reader imagine a completed system.

A key requirement of aggregate creatives is flexibility. Each type of container has strict needs for formatting HTML and controlling the number and types of sub-creatives that appear within it.

The flipside to flexibility is complexity. We have decided to put the complexity of aggregate creatives into the backend administration tools in hands of the Rich Media team. Another choice would have been in the ad calls themselves owned by Ad Production, but that approach complicates run-time processing and cannot be designed as flexibly as a back-end solution.

Figure 1-3 – Architecture of Aggregate Creatives



The overall architecture shown in Figure 1-3 assumes an important design decision based on a strict performance requirement: we pre-generate all possible forms of an aggregate creative offline rather than attempt to dynamically determine the form on a busy ad engine.

1.2.1 Staging

A critical set of requirements for operational success of the aggregate feature is the ability to verify complex containers. This should be set up in a way such that other types of creatives beyond aggregates can benefit. The architecture of the staging environment is:

1. Set a stage-status flag on the Line in the CMI
2. Production environment ignores Lines flagged this way.

3. A special staging environment treats staged lines as if they were active regardless of date. (i.e. a Line marked as staged will deliver before and after its end date). This implies there will be separately-configurable dashboards. That in turn implies that a new table is required: StagedLineRotatorDistribution.

1.4 Aggregate Creatives

Benefits: - Reporting by separate orders
- One container needed per opportunity
- Basic automated rotation
- Unlimited formatting features

Drawbacks: - Initially, percentage-based only.
- Initially, single-tier only

The aggregate creative containers must be a general purpose programming language with several added routines in support of the specific requirements.

1.4.1 Sample Commands

This section illustrates the requirements that would be placed upon commands and parameters in an initial implementation. The commands shown are for example purposes only. The final syntax may vary.

```
command=createPool
        pool=[name]         ; defines name for future support of multiple pools
        target=[targetId]   ; opportunity target ID
        seats=[number]      ; number of seats to be drawn from pool
command=getSegment
        pool=[name]             ; defines the pool to draw from
command=getPoolSize

command=<init>, </init>         ; define the initialization section
```

1.4.2 Rotation

The algorithm for handling generation of two pools of different sizes is to slide once along the longest one, while sliding along the shorter one as many times as necessary. e.g.

```
Pool-1=A,B,C,D      ; 2 seats
Pool-2=1,2,3        ; 2 seats

Result is AB-12, BC-23, CD-31, DA-12
```

As opposed to: AB-12,AB-23,AB-31,BC-12,BC-23,BC-31
CD-12,CD-23,CD-31,DA-12,DA-23,DA-31

For pools of large size, this method could save significant processing time.

1.4.3 Example

Below is an example programmed container with the following functionality:

1. Set up 2 named pools of sub-creatives using a targetId
2. Display an arbitrary number of sub-creatives from the pool
3. Rotate through the pool with a *shifting* algorithm, e.g: 1-2-3, 2-3-1, 3-1-2.

Percentage (share) based targeting only - no impression goals.

```
# declare my 2 pools

# pool 1: button bar; 3 seats
my $buttonPool = createPool("buttonPool", 123, 3);

# pool 2: link list; 2 seats
my $linkPool = createPool("linkPool", 789, 2);

__BEGIN_BODY__

# the tracking GIF
addText("\${TRACK:PIXEL} <!-- tracking gif for the container -->\n");

# get my seat assignments
my $button1 = getSegment("buttonPool");
my $button2 = getSegment("buttonPool");
my $button3 = getSegment("buttonPool");
my $link1 = getSegment("linkPool");
my $link2 = getSegment("linkPool");

my $buttonPoolSize = getPoolSize("buttonPool");

# output the buttons
my $text = qq{
<table border='1'> <!-- outer table -->
};

if ($buttonPoolSize > 0) {
        $text .= "<table> <!-- button bar table-->";

        if ($buttonPoolSize >= 3) {
                $text .= "<td> $button1 </td>";
        }
        $text .= "<td> $button2 </td>";

        if ($buttonPoolSize >= 2) {
                $text .= "<td> $button3 </td>";
        }

        # close the table
        $text .= "</table> <!-- /button bar table -->";
}
else {
        $text .= "<!-- no buttons scheduled -->  ";
}
```

```
addText($text);

# print a horizontal rule, only if 2nd pool not empty
if (getPoolSize("linkPool") > 0) {
        addText("\n<hr>\n\n");
}


# ???
addText("<h5>Enjoy these 2 of " . getPoolSize("linkPool") . " sponsored
        links:</h5>");

# output the links
$text = qq{
   <ul>
      <li> $link1 </li>
      <li> $link2 </li>
   </ul>

</table> <!-- /outer table -->
};

addText($text);

# done!
```

1.4.4 Detailed Requirements

Specific requirements for macros, in addition to the general requirements noted in Section 2:

R1-1: Aggregate Creatives must support 2 pools, and should support more.
R1-2: Aggregates must support weighted segments for percentage-based opportunities
R1-3: Aggregates should support impression-based opportunities
R1-4: Unsold opportunities must be able to gracefully degrade to overall default HTML for the container
R1-5: Required and default constraints are: notSameSegment, notSameCreative, notSameLine. A 'notSameAdvertiser' constraint should be supported on a per-pool basis.
R1-6: The system should allow for graceful formatting of 'short pools' – where the number of sold opportunities is less than the number of seats.
R1-7: The order of the pools must be randomized before picking sub-creatives

1.6 Targetable Areas

Ideally, Aggregate Creatives could be structured to run in Run-of-Channel and Run-of-Brand rotations covering several smaller targetable areas.

However, it is recommend that they be initially limited to one targetable area (rotator), with no run-of-channel type containers. Channel-level opportunities are fine, so long as they cannot be broken down into targetable sub-channels. (i.e. limit one rotator)

R1-8: The system *should* support multi-tier opportunities.

Which means initial implementations probably won't support more than a single tier. Why? A major difference between Aggregate and "flat" creatives is *dependent rotators*. The opportunity's target-tree shadows the container's target-tree, but only the container's rotators have mapping files. The opportunity's rotators will never normally get called directly by the page… their delivery depends on the delivery of the container's rotator.

To correctly track complex tiering, additional tracking maps would be needed at runtime. For example:

Table 1-1 – Sample Dependent Rotator Map

	Container Rotator	Link Rotator	Button Rotator
DL:GEN	100	197	288
DL:UTILITIES	101	224	292
DL:HOME	102	231	301
DL:GAMES	103	265	322

2.0 General Requirements

Two types of detailed requirements: *must* (mandatory) and *should* (optional).

General

R2-1: A method must exist to test aggregate creatives, whether static or dynamic.

This means developing a staging system.

R2-2: Updates to sub-creatives should be automatically propagated to all containers that include them.

Tracking

R2-3: The system must support special tracking requirements for sub-creatives such that advertisers may receive online real-time delivery reports. To make this happen, the ${TRACK:*} variables embedded in the sub-creatives must have SEGMENT-ID and RGROUP resolved. RGROUP may be resolved to a variable for runtime dependent rotator mapping.

R2-4: The system must produce all support elements: html, js, ifc, wc. In addition, note that aggregates may be wrapped by an iframe.

Variable Resolution

R2-5: The system should support full variable resolution syntax.

Weights

R2-6: For dynamic creatives (macros and programmable), it should be possible to defined the relative weights of sub-creatives.

Constraints

R2-7: For dynamic creatives, it must be possible to define constraints affecting which sub-creatives are chosen.

Administration

R2-8: It should be convenient to edit aggregate creatives, whether static or dynamic, in an external editing tool.

Runtime support

R2-9: The runtime delivery environment must support the choosing from multiple forms of the same segment.

R2-10: The runtime environment should support the mapping of dependent rotators from page rotators.

Performance

R2-11: Each static aggregate creative should add no more than 5 seconds to offline processing. Macro creatives should add no more than 10 seconds each on average. The system should log warnings when a creative's processing exceeds a configurable limit (10 seconds). Scale the system for 100 Macros, average of 1 pool, 20 creatives, and 4 seats.

R2-12: At runtime, the effect of processing extra creative forms and runtime variables should be no more than 5ms latency at full throughput.

EXHIBIT D

CVS log for ad/docs/madison/aggregate_creative_reqs.doc

Up to [CNET] / ad / docs / madison

Request diff between arbitrary revisions

Keyword substitution: b (i.e.: CVS considers this a binary file)
Default branch: MAIN

Revision **1.5**: download
Fri Sep 6 21:59:40 2002 UTC (6 years, 4 months ago) by *bretg*
Branches: MAIN
CVS tags: HEAD
Changes since revision 1.4: +50 -115 lines

```
update for Perl creatives, staging details
```

Revision **1.4**: download
Mon Aug 19 23:11:49 2002 UTC (6 years, 5 months ago) by *bretg*
Branches: MAIN
Changes since revision 1.3: +48 -64 lines

```
reviewed with addev, removed change bars
```

Revision **1.3**: download
Mon Aug 19 14:20:51 2002 UTC (6 years, 5 months ago) by *bretg*
Branches: MAIN
Changes since revision 1.2: +110 -74 lines

```
added macro aggregates
```

Revision **1.2**: download
Thu Aug 8 14:05:29 2002 UTC (6 years, 5 months ago) by *bretg*
Branches: MAIN
Changes since revision 1.1: +28 -39 lines

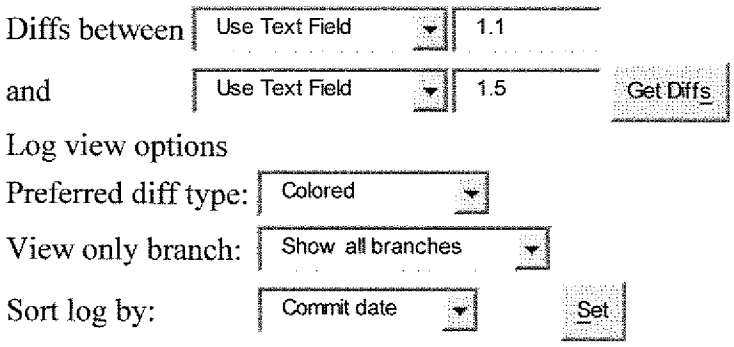
```
fleshed out requirements, API
```

Revision **1.1**: download
Wed Aug 7 22:33:18 2002 UTC (6 years, 5 months ago) by *bretg*
Branches: MAIN

```
initial control
```

Diff request

This form allows you to request diffs between any two revisions of a file. You may select a symbolic revision name using the selection box or you may type in a numeric name using the type-in text box.



FreeBSD-CVSweb <freebsd-cvsweb@FreeBSD.org>